Fixed-Point Trigonometric Functions on FPGAs

Florent de Dinechin, Matei Istoan, Guillaume Sergent

Fixed-Point Trigonometric Functions on FPGAs

Florent de Dinechin, Matei Istoan, Guillaume Sergent
AriC project, LIP (ENSL-CNRS-INRIA-UCBL), Université de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France
{Florent.de.Dinechin, Matei.Istoan, Guillaume.Sergent}@ens-lyon.fr

ABSTRACT

Three approaches for computing sines and cosines on FPGAs are studied in this paper, with a focus of high-throughput pipelined architecture, and state-of-the-art implementation techniques. The first approach is the classical CORDIC iteration, for which we suggest a reduced iteration technique and fine optimizations in datapath width and latency. The second is an ad-hoc architecture specifically designed around trigonometric identities. The third uses a generic table- and DSP-based polynomial approximator. These three architectures are implemented and compared in the FloPoCo framework.

1. INTRODUCTION

Sine and cosine functions are quite pervasive in signal processing applications. We are interested here in comparing the efficiency on FPGA of several approaches that compute the sine and/or the cosine of a number in fixed point.

Some application only require the sine or the cosine of a value at some point, but a fused sine-and-cosine implementation is interesting for several reasons. First, many applications require both, for instance to implement rotations. Second, many methods compute both anyway. Indeed, most such methods are derived from the following identity on complex exponential:

$$e^{j(a+b)} = e^{ja} \times e^{jb}$$

or its real version, which is used in practice:

$$\begin{cases} \sin(a+z) = \sin(a)\cos(z) + \cos(a)\sin(z) \\ \cos(a+z) = \cos(a)\cos(z) - \sin(a)\sin(z) \end{cases} \quad (1)$$

These equations are best understood as follows: a rotation of angle $a+b$ is obtained as the composition of the rotation of angle a and the rotation of angle b. This identity can be applied several times, and the sine and cosine of a given angle α are eventually obtained by rotating the complex point 1 by the angle α.

Overview of existing algorithms

The well known CORDIC algorithm, due to Volder [11], is a classical technique to compute both sine and cosine using such a decomposition into micro-rotations. These rotations are chosen in such a way that (1) can be computed only using additions and shifts. CORDIC has been present in vendor core generators almost from the start, well studied on FPGAs [1], and still, FPGA implementations are submitted every year to FPGA conferences, with very little or no novelty compared to the original article by Volder. However, many CORDIC variations have been designed over time, see for instance Muller's textbook [9] or an excellent review written for the 50 years of CORDIC [8]. However, mostly due to the fact that standard additions are accelerated by fast-carry logic on FPGAs, the preferred implementation remains the simplest one. Roughly speaking, to compute w-bit sine and cosine in a pipeline fashion, it requires $3(w+g)$ additions of size $w+g$ where $g = \lceil log_2 w \rceil + 1$ is a number of guard bits ensuring last-bit accuracy.

FPGAs were then enhanced with embedded multipliers, then DSP blocks. These resources can also be used to evaluate sine and cosine, using Equation (1) with a coarser decomposition of an input angle into a sum of two smaller angles. A good decomposition of α is $\alpha = \alpha_h + \alpha_l$ where α_h is the number formed from the k leading bits of α, and α_l is formed of the remaining lower bits, so that $\alpha_l < 2^{-k}$. This enables the use of an acceptable table for $\sin(\alpha_h)$ and $\cos(\alpha_h)$, and a small degree polynomial, typically Taylor, for $\sin(\alpha_l)$ and $\cos(\alpha_l)$. This idea will be exploited in Section 4. However, other decompositions have been suggested [7, 9].

A variant of CORDIC, called "reduced iterations CORDIC" [10], uses a similar idea. It first rotates the w-bit input angle using roughly $w/2$ CORDIC microrotation. The remaining angle z is then smaller than $2^{-w/2}$. Its sine and cosine can be approximated (with good enough accuracy) as $\sin z \approx z$ and $\cos z \approx 1$. In both cases the approximation error is smaller than 2^{-w}. This ensures that a final rotation of angle z can be computed, using (1), with just two small multiplications (inputs of size roughly $w/2$) and two additions.

The purpose of this article is to survey all these techniques for the specific context of modern FPGAs with embedded memories, embedded DSP blocks, and large LUTs.

Specification

In this paper we compute a fixed-point approximation of $\sin(\pi x)$ and $\cos(\pi x)$, where x is a signed (two's complement) number on w bits in $[-1, 1)$. This specification is quite natural, as it maps the modulo-2π trigonometric periodicity to the modulo-2 reduction behind the two's complement representation. In other words, using such an implementation, we can compute sines and cosines of numbers in a wider fixed-point range, for instance having k integer bits, by simply dropping these bits: this corresponds to the periodicity of the sine.

However, the range $[-1, 1)$ is not symmetric in two's complement fixed-point: -1 is representable, but not 1. On the input side, this asymmetry is not a problem, it actually matches well the periodicity of the input. On the output side, however, there is, classically, a small problem: we want a fixed-point result on n bits, so if we want to avoid overflow, the value 1 must be actually represented as $0.11..11_2 = 1 - 2^{-w}$. By symmetry, the output value -1 should be represented as $-1+2^{-w}$, not as the (representable) -1.

Therefore, what an w-bit trigonometric operator should compute is the function $(1 - 2^{-w})\sin(\pi x)$ or $(1 - 2^{-w})\cos(\pi x)$. This is not a problem in practice. First, applications of this operator can always reduce the corresponding systematic error by increasing the precision w. Second, the sequel will show that for all proposed architectures, the scaling factor $(1 - 2^{-w})$ can be computed at no cost.

The implementations described in this article all provide the same numerical quality: all the operators studied in the following are *faithfully rounded* (another wording is *last-bit accurate*): they return a value whose error with respect to the true mathematical value is provably smaller than the weight of the least-significant bit of the result. This ensures that all the bits of the result are useful. Indeed, any error larger than that would mean that we output useless bits, which would waste routing resources [5]. All the synthesis results correspond to architectures that have been extensively tested against this accuracy specification.

We focus on fixed-point for two reasons. Firstly, with its output in $[-1, 1]$ and the periodicity property on its input, a trigonometric function should, in most applications, be considered a fixed-point object. In other words, we claim that most applications involving a floating-point sine or cosine will benefit, when implemented on FPGAs, from a fixed-point conversion of the datapath around these functions. Secondly, we have studied in detail elsewhere [6] the overhead of accurate floating-point implementations of sine and cosine. Most of the present work is also relevant to such floating-point implementations.

Finally, we focus on fully pipelined architectures able to produce one result per cycle. Two of the three techniques studied here could lead to sequential implementations that take several cycles to compute one result, but these are out of scope of the present article.

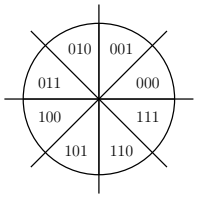
Outline and contributions

Section 2 will overview the common (and classical) argument reductions that benefit to the three techniques presented in this article. Section 3 focuses on CORDIC for FPGAs. Several minor contributions (a proper error analysis, a flexible pipeline, a reduction of some datapath widths, an implementation of the reduced iteration technique) sum up to a state-of-the art implementation of unrolled CORDIC. Section 4 presents an original rotation architecture targeting modern FPGAs. It is based on tables (matching embedded RAMs) and multiplications (matching embedded DSP blocks). The choice of its various parameters is discussed. Section 5 presents another table- and multiplier-based architecture, this time computing only one of sine or cosine. It uses a generic polynomial evaluation technique. Section 6 compares synthesis results for a range of precisions and a range of frequency targets.

A final contribution is the availability of open-source, high quality implementations of these operators in FloPoCo. All the presented operators are available for evaluation in FloPoCo's public subversion repository at submission date. They will be included in future FloPoCo releases.

2. ARGUMENT REDUCTION

The first argument reduction (very classically) considers the three leading bits of x, which we call s (for sign), q (for quadrant) and o (for octant). We call the remaining bits $y \in \left[0, \frac{1}{4}\right)$. We now may use classical trigonometric identities such as $\sin(\frac{\pi}{2} - x) = \cos(x)$ to reduce the problem to computing $\sin(\pi y')$ and $\cos(\pi y')$ for $y' \in \left[0, \frac{1}{4}\right)$.



$$y' = \begin{cases} \frac{1}{4} - y \text{ if } o = 1 \\ y \text{ otherwise.} \end{cases}$$

sqo	Reconstruction
000	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = \cos(\pi y') \end{cases}$
001	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = \sin(\pi y') \end{cases}$
010	$\begin{cases} \sin(\pi x) = \cos(\pi y') \\ \cos(\pi x) = -\sin(\pi y') \end{cases}$
011	$\begin{cases} \sin(\pi x) = \sin(\pi y') \\ \cos(\pi x) = -\cos(\pi y') \end{cases}$

Figure 1: Argument reduction as a function of the (s, q, o) bits (cases $s = 1$ not shown)

3. VARIATIONS ON CORDIC

The classical CORDIC iteration for computing sine and cosine is the following:

$$\begin{cases} c_0 &= \frac{1}{\prod_{i=1}^{n}\sqrt{1+2^{-i}}} \\ s_0 &= 0 \\ z_0 &= y \quad \text{(the reduced argument)} \end{cases} \tag{2}$$

$$\begin{cases} d_i &= +1 \text{ if } z_i > 0, \text{ otherwise } -1 \\ c_{i+1} &= c_i - 2^{-i} d_i s_i \\ s_{i+1} &= s_i + 2^{-i} d_i c_i \\ z_{i+1} &= z_i - d_i \arctan(2^{-i}) \end{cases} \tag{3}$$

Here c_i and s_i converge to the cosine and sine of the input y [9].

We implemented in FloPoCo a generator of such CORDIC rotator, in unrolled (fully pipelined) fashion. In this case, the values of $\arctan(2^{-i})$ are constant inputs to the additions. Equation (3) works for radian arguments, but adapting it to our specification is simply a matter of scaling each constant $\arctan(2^{-i})$, so it entails no cost. Similarly, the $1 - 2^{-w}$ scaling is merged at no cost in the initial scaling factor of Equation (2). In this equation, n is the number of iterations, slightly larger than w – more on this below.

3.1 Error analysis

To ensure last-bit accuracy at the lowest possible cost, we need to analyze the errors performed by each iteration. First, the method error entailed by Equation (3) is of the order of the value of z_i when we stop the iteration. Its bound can be computed numerically, and the iteration count n is defined as the smallest integer for which this method error is smaller than 2^{-w-2}.

Then, we have to bound the rounding errors. All the computations are performed on $w+g$ bits, where g is a number of guard bits (to be determined now) that will absorb the accumulation of these errors. Let us call $u = 2^{-w-g}$ the value of the unit in the last place on this datapath.

On the z_i datapath, each iteration adds a constant that is the correct rounding of the actual $\arctan(2^{-i})$ to the precision u. This adds an absolute error bounded by $u/2 = 2^{-w-g-1}$ to the error of the previous iteration, so the accumulated error after i iterations is bounded by $i.u/2$.

On the c_i and s_i datapath, we may express an error bound ϵ_i (expressed in u) for iteration i by the following recurrence:

$$\left\{ \begin{array}{rcl} \epsilon_0 & = & 0 \\ \epsilon_{i+1} & = & (\epsilon_i + 2^{-i}\epsilon_i + 1)u \end{array} \right. \quad (4)$$

where the $+1$ comes from the truncation of the shifted addend (e.g. $2^{-i}d_ic_i$). Note that this term could be reduced to $1/2$ by rounding the shifted addend instead of truncating it. However this would require adders larger by one bit, eventually dividing the overall error by less than two, so saving at most one iteration. If the CORDIC operator is viewed as a square of full adders, we would win one bit on one size of the square and lose it on the other side: there is no net gain.

One observes that the rounding error bound on c_i and s_i is always larger than that on z_i. Our implementation therefore computes ϵ_n numerically and uses it to determine the smallest g ensuring $\epsilon_n.2^{-w-g} < 2^{-w-2}$.

The sum of the method error and the rounding error is thus smaller than 2^{-w-1}. The rounding of c_n and s_n to the target precision 2^{-w} entails another error bounded by 2^{-w-1}. Thus, the overall error is strictly smaller than 2^{-w}, ensuring the faithful rounding property.

3.2 Reducing the z_i datapath

Another minor contribution of our implementation is to compute just right the iteration on z. Indeed, all this computation is implemented in fixed point, and it can be observed that each iteration roughly removes one most significant bit to the angle z_i. In other words, z_i needs only be computed on $w+g-i$ bits. This reduces the total area by about 1/6th.

In principle it could also slightly reduces the critical path: in the early iterations, the critical data dependency of one iteration to the next one is $z_i \rightarrow d_i \rightarrow z_{i+1}$. If we know d_i earlier because z_i is shorter, we can start an iteration on c_{i+1} and s_{i+1} before the end of the previous one: as soon as the bits of c_i and s_i that are needed for c_{i+1} and s_{i+1}, we can launch the computation of c_{i+1} and s_{i+1}, and the carry propagations of the two additions will execute in parallel. In the later iterations, as c_i and s_i are being shifted further, the gain is smaller, which is unfortunate because that is when d_i can be computed the fastest. We are currently trying to express these subtleties in FloPoCo's pipelining framework to reduce the latency of the CORDIC implementation.

We observe that current synthesis tools are able to pack the c_i and s_i lines of Equation (3) as one LUT per bit, while still using the fast-carry line. The LUT consumption reported in Table 3 is thus very predictable, close to $2.5(w+g)^2$.

3.3 Reduced iterations CORDIC

In this version, the CORDIC iteration is stopped as soon as the remaining rotation can be computed, with sufficient accuracy, by

$$\left\{ \begin{array}{l} x_{i+1} = x_i + zy_i \\ y_{i+1} = y_i - zx_i \end{array} \right. \quad (5)$$

The size of z is slightly more than $w/2$. We may truncate x_i and y_i before the products to the same accuracy, with only an additional contribution of 2^{-w-g} to the overall error. Thus, only two multipliers of roughly $w/2 \times w/2$ bits are needed. In practice, the reduced iteration approach will consume only two of 18×18-bit signed multipliers of DSP-enabled FPGAs for values of w up to 32.

The computation of the initial scaling factor must be modified accordingly, and other technical details can be found in the FloPoCo code.

4. A TABLE- AND DSP-BASED PARALLEL POLYNOMIAL ARCHITECTURE

4.1 Algorithm

Here we further split our octant angle y into its a most significant bits t, and its lower bits $y_{\text{red}} \in [0, 2^{-(2+a)})$. We use t to address a table storing $sin(\pi t)$ and $cos(\pi t)$. Meanwhile, the sine and cosine of πy_{red} are evaluated by first computing $z = \pi y_{\text{red}}$, then by using the Taylor series of $\sin z$ and $\cos z$.

We chose Taylor series over generic polynomial approximation as in [4], for two reasons. The first is that the sine series is odd and the cosine series is even, so their evaluation requires half as many multiplications for a given degree. The second is that up to terms of degree 4, the coefficients are powers of two, or powers of two multiplied by 1/3. Powers of two are for free, and division by 3 is very cheap on FPGAs [2].

In this implementation we choose a such that $4(a+2)-2 \geq w+g$ which implies that $\frac{z^4}{24} \leq 1.02 \times 2^{-(w+g)}$.

It enables us to neglect terms beyond $\frac{z^3}{6}$, yielding the following formulae:

$$\begin{array}{rcl} \sin z & \approx & z - \dfrac{z^3}{6} \\ \cos z & \approx & 1 - \dfrac{z^2}{2} \end{array}$$

Once we have both $\sin(\pi t)$, $\cos(\pi t)$ and $\sin(\pi y_{\text{red}})$, $\cos(\pi y_{\text{red}})$, we can now use the addition formulae to recover the values of $\sin(\pi y_{\text{in}})$ and $\cos(\pi y_{\text{in}})$:

$$\begin{array}{rcl} \sin(\pi y_{\text{in}}) & = & \sin(\pi t)\cos z + \cos(\pi t)\sin z \\ \cos(\pi y_{\text{in}}) & = & \cos(\pi t)\cos z - \sin(\pi t)\sin z \end{array}$$

Again, this is the rotation equation (1). Actually, $\cos z$ is always very close to 1 so computing $\sin(\pi t)\cos z$ (for example) is significantly costlier than computing $\sin(\pi t) + \sin(\pi t)(\cos z - 1)$. Therfore, instead of the above addition formulae we use the following ones:

$$\begin{array}{rcl} \sin(\pi y_{\text{in}}) & = & \sin(\pi t) - \sin(\pi t)\dfrac{z^2}{2} + \cos(\pi t)\sin z \\ \cos(\pi y_{\text{in}}) & = & \cos(\pi t) - \cos(\pi t)\dfrac{z^2}{2} - \sin(\pi t)\sin z \end{array}$$

4.2 Implementation details

$\frac{1}{4} - y$ is computed as $\neg y$, inducing an error of $2^{-(w+g)}$. This avoids one overflow situation and saves a carry propagation.

w	a	y_{red}, z, cosAsinZ, sinAsinZ	$z^2/2$	$z^3/6$
16	4	16	11	6
24	6	22	15	8
32	8	28	19	10
40	10	34	23	12
48	12	40	27	14

Table 1: Internal precisions needed by the architecture of Figure 2. The $z^2/2$ column also defines the size of the two left multipliers on the figure. The y_{red} column defines the size of the two rightmost multipliers.

The corresponding architecture is depicted on Figure 2. Before each multiplication, we truncate the two inputs to the precision of the result: this is illustrated by the small boxes labelled T. We also use truncated multipliers, and $z^2/2$ is computed using a squarer. Table 1 shows the internal precisions used on this figure for various value of the input/output precision w.

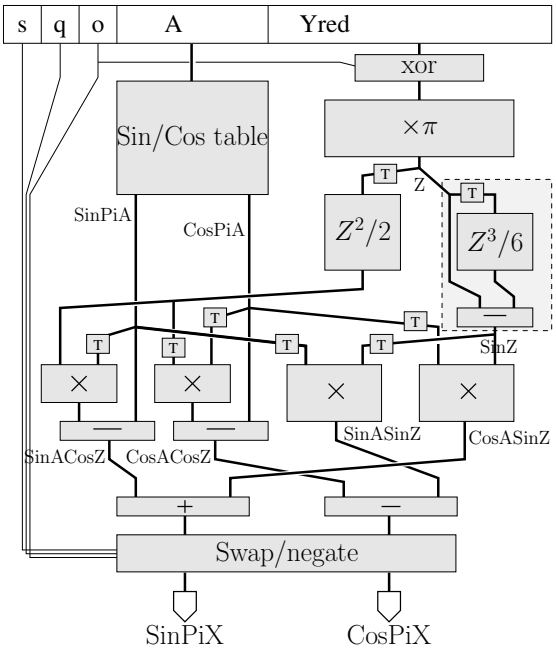


Figure 2: A DSP- and table-based architecture

As we need very few bits of $z^3/6$, this value can be simply tabulated for small sizes of $z^3/6$ (currently up to 10 bits). Otherwise, we designed an ad-hoc architecture that computes $z - z^3/6$ (the dashed box of Figure 2) inside a single bit heap [3]. Only a few bits need to be divided by 3, using a variation of [2]. An example of bit heap thus obtained is shown on Figure 3.

4.3 Error analysis

We again have a fixed-point architecture which loses up to 15 ulps to rounding/truncation errors: one half-ulp for each table, one ulp for each truncation T and for each truncated multiplier or squarer, and one ulp for the initial computation of $1/4 - y$ by xoring. Therefore, the value of g that enables last-bit accuracy is 4 – it no longer depends on w. The data

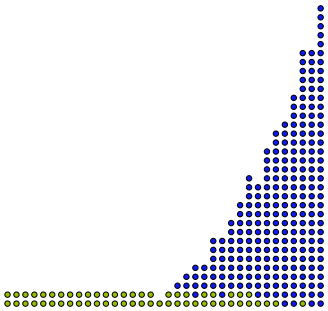


Figure 3: The bit heap computing $z - z^3/6$ for a 40-bit sine/cosine operator.

in Table 1 includes these guard bits.

5. ARCHITECTURE USING A GENERIC POLYNOMIAL EVALUATOR

The last option we have explored is a wrapper of the generic polynomial evaluator presented in [4] in the range reduction of Section 2.

In addition to the argument x, this operator inputs a bit $\overline{\sin\cos}$ that determines if the sine or the cosine should be computed. Here x is only reduced to one quadrant: $y' \in [0, 1/2)$. The polynomial evaluator computes $\sin(\pi y')$ for $y' \in [0, 1/2)$. The proper output is reconstructed out of $\sin(\pi y')$ and the bits s, q and $\overline{\sin\cos}$.

6. RESULTS AND DISCUSSION

In Table 3, SinAndCos denotes the combined sine/cosine operator presented in Section 4, SinOrCos is the single-output operator of Section 5 for which d is the degree of the polynomial approximation. All the results are for Virtex-5(5vfx100tff1738-3), very comparable results can be obtained for other Xilinx or Altera targets. We can obtain various frequency/latency trade-offs by varying the `-frequency` option to FloPoCo.

The CORDIC results match those reported for the corresponding Xilinx logicore operator [12] in parallel mode. The reduced iteration approach is very attractive, halving the iteration count at the cost of two DSP blocks only for up to 32 bits.

A first question one may ask is: on an FPGA without DSP blocks, do the multiplicative approaches makes sense? A partial answer is obtained by comparing in Table 2 a combinatorial CORDIC, and a LUT-based combinatorial version of the operator of Figure 2, where the tables and the multipliers are implemented using LUTs only. These two implementations are functionally equivalent (same inputs, same outputs, same accuracy). For the soft multipliers, we use the FloPoCo implementation of truncated soft multipliers in order to benefit from truncation. We observe that the two approaches consume comparable resources. However, the multiplier-based approach drastically reduces the latency. We conclude that there is little point in using unrolled CORDIC in this context. However, CORDIC is still relevant in its iterative implementation (there is no iterative version of the operator of Figure 2).

The DSP-based approaches are increasingly relevant as

precision increases. Their frequency doesn't match yet that of the CORDIC operators. This is not due to an intrinsic limitation of the algorithms used, but to the more complex pipeline construction. The current FloPoCo framework is not very good at exploiting the internal registers of DSP blocks. This is being worked upon. All the DSP-based operators should be able to reach 400 MHz on Virtex-5, albeit at the cost of a longer latency and more registers.

When comparing SinAndCos and SinOrCos, one should remember that it takes two instances of the second to emulate the first. SinAndCos is therefore more efficient than SinOrCos, although not by a very large margin.

7. CONCLUSION AND PERSPECTIVES

This article is a survey of sine and cosine evaluation on modern FPGAs with DSPs and embedded memories. It evaluates two relevant variations on the classical CORDIC, and two DSP-based techniques. This work is probably the state of the art, although we still expect performance improvements. At any rate it exposes a lot of trade-offs between performance (frequency and latency) and resource consumption (logic, DSP, memories).

Short-term future work includes more tuning. Floating-point versions of the trigonometric functions should also be provided, using similar techniques [6].

The technique presented here will scale well to double precision. However, the ROM size will then becomes a problem. An obvious solution is to decompose the input further and compose more rotations, at the cost of more multiplications. An alternative approach is to address the table with fewer bits, but allow larger degree polynomials for $\sin z$ and $\cos z$, which also costs more multiplications. Which solution wins is not obvious.

As such questions illustrate, there are more fundamental questions behind the search for efficient architectures on today's FPGAs: How many bits does one need to flip and move in order to compute a sine faithful to w bits, and at what hardware cost? Asking this question properly requires to model the compared costs of FPGA logic, embedded memories, DSP blocks, etc. Then, such a question is essentially answered by exhibiting and comparing architectures, as we have done in this article.

Table 2: Synthesis results on Virtex-5 for logic-only implementations.

Approach	latency	area
CORDIC 16 bits	30.3 ns	1034 LUTs
SinAndCos 16 bits	15.0 ns	1211 LUTs
CORDIC 24 bits	44.6 ns	2079 LUTs
SinAndCos 24 bits	17.0 ns	2183 LUTs
CORDIC 24 bits	62.1 ns	3513 LUTs
SinAndCos 24 bits	19.4 ns	3539 LUTs

8. REFERENCES

[1] R. Andraka. A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200. ACM, 1998.

[2] F. de Dinechin and L.-S. Didier. Table-based division by small integer constants. In *Applied Reconfigurable Computing*, pages 53–63, Hong Kong, Mar. 2012.

[3] F. De Dinechin, M. Istoan, G. Sergent, K. Illyes, B. Popa, and N. Brunie. Arithmetic around the bit heap. Technical report, Oct. 2012.

[4] F. de Dinechin, M. Joldes, and B. Pasca. Automatic generation of polynomial-based hardware architectures for function evaluation. In *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

[5] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[6] J. Detrey and F. de Dinechin. Floating-point trigonometric functions for FPGAs. In *Field-Programmable Logic and Applications*, pages 29–34. IEEE, 2007.

[7] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.

[8] P. K. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna. 50 years of cordic: Algorithms, architectures, and applications. *IEEE Transactions on Circuits and Systems I : Regular papers*, 56(9):1893–1907, Sept. 2009.

[9] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 2nd edition, 2006.

[10] D. Timmermann, H. Hahn, and B. Hostika. Modified CORDIC algorithm with reduced iterations. *Electronics Letters*, 25(15):950–951, 1989.

[11] J. Volder. The CORDIC trigonometric computing technique. *IRE Trans. Electronic Computing*, EC-8:330–334, Sept.

[12] Xilinx Corporation. *CORDIC v4.0 (DSD249)*, 2009.

Table 3: Comparison of the three techniques on Virtex-5.

Approach	latency	frequency	Reg. + LUTs	BRAM	DSP
precision = 16 bits					
CORDIC	18	478	969 + 1131	0	0
CORDIC	14	277	776 + 1086	0	0
CORDIC	7	194	418 + 1099	0	0
CORDIC	3	97	262 + 1221	0	0
Red. CORDIC	13	368	625 + 719	0	2
Red. CORDIC	4	238	106 + 713	0	2
SinAndCos	4	298	107 + 297	0	5
SinAndCos	3	114	168 + 650	0	2
SinOrCos (d=2)	9	251	136 + 183	1	2
precision = 24 bits					
CORDIC	28	439.9	1996 + 2144	0	0
CORDIC	23	251.0	1721 + 2114	0	0
CORDIC	12	180.5	919 + 2146	0	0
CORDIC	5	103.8	442 + 2089	0	0
Red. CORDIC	20	273.4	1401 + 1446	0	4
Red. CORDIC	11	222.6	674 + 1438	0	4
Red. CORDIC	5	222.6	224 + 1470	0	2
SinAndCos	5	262	197 + 441	3	7
SinAndCos	3	179	193 + 472	1	7
SinOrCos (d=2)	9	251	202 + 279	2	2
SinOrCos (d=2)	7	180	178 + 278	2	2
precision = 32 bits					
CORDIC	37	403.5	3495 + 3591	0	0
CORDIC	32	230.0	3120 + 3559	0	0
CORDIC	16	162.4	1532 + 3509	0	0
CORDIC	7	86.6	698 + 3508	0	0
Red. CORDIC	24	256.8	2160 + 2234	0	4
Red. CORDIC	15	217.1	1149 + 2295	0	4
Red. CORDIC	7	112.1	618 + 2225	0	4
SinAndCos	10	253	535 + 789	3	9
SinAndCos	4	110	311 + 996	1	9
SinOrCos (d=3)	14	251	444 + 536	4	5
SinOrCos (d=3)	9	146	306 + 534	4	5
precision = 40 bits					
CORDIC	45	375	5070 + 5289	0	0
CORDIC	25	149	2948 + 5245	0	0
Red. CORDIC	37	252	3695 + 3768	0	8
Red. CORDIC	22	211	2438 + 3476	0	8
Red. CORDIC	9	123	931 + 3339	0	8
SinAndCos (bit heap)	11	266	895 + 1644	3	12
SinAndCos (table $z^3/6$)	8	232	500 + 949	4	12
SinAndCos (bit heap)	4	154	612 + 2826	0	12
SinAndCos (table $z^3/6$)	4	156	395 + 2268	2	12
SinOrCos (d=3)	15	251	628 +725	4	8
SinOrCos (d=3)	9	132	376 +675	4	8
precision = 48 bits					
SinAndCos (bit heap)	13	232	1322 + 2369	12	17
SinAndCos (bit heap)	6	132	972 + 2133	12	17
SinOrCos	15	250	734 + 879	17	10
SinOrCos	9	124	431 + 823	17	10